Dshell++: A Component Based, Reusable Space System Simulation Framework

Christopher S. Lim and Abhinandan Jain
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, Pasadena, CA 91009 USA
Email: Christopher.S.Lim@jpl.nasa.gov

***Abstract*—This paper describes the multi-mission Dshell++ simulation framework for high fidelity, physics-based simulation of spacecraft, robotic manipulation and mobility systems. Dshell++ is a C++/Python library which uses modern script-driven object-oriented techniques to allow component reuse and a dynamic run-time interface for complex, high-fidelity simulation of spacecraft and robotic systems. The goal of the Dshell++ architecture is to manage the inherent complexity of physics-based simulations while supporting component model reuse across missions. The framework provides several features that support a large degree of simulation configurability and usability.**

***Index Terms*—Aerospace simulation software**

I. Introduction

Traditionally, spacecraft simulations have been built as a monolithic program targeted to a specific mission application [1]. Early simulators were developed using a procedural, non-object-oriented programming language optimized for mathematical computations such as FORTRAN [1]. Adding or changing a feature, altering the internal data structures or even modifying the format of the input/output data usually required a deep understanding of the entire simulation [2]. With the advent of personal computers and workstations, tools such as Matlab and Simulink [3] gained popularity for building simulations. These tools introduced a visual, interactive interface but lack the capability to work well in embedded systems which required real-time performance.

Dshell++ is a high fidelity, multi-mission, physics-based simulation toolkit with the goal of increasing simulation productivity by using modern object-oriented techniques to allow component reuse, a data-flow architecture, and a dynamic run-time interface for complex, high-fidelity spacecraft and robotics systems simulations.

Object-Oriented Design

Dshell++ is the next generation version of the Dshell spacecraft dynamics simulator [4] completely redesigned and rewritten in C++. Dshell++ uses object-oriented techniques to allow code reuse and component building. Dshell++ simulations consist of a collection of component device models from model libraries assembled and connected together into a data flow to meet the required simulation behavior. For example, a thruster model is a C++ class derived from (and inherits all the properties of) an actuator C++ base class which models a device that applies a force on a body. Dshell++ provides facilities for the inter-connection and efficient data exchange between such models. Related models within sub-systems (for example, a bank of thrusters) can be grouped together into an "assembly" which in turn can be part of a larger assembly. The assemblies are reusable and can be used across more than one simulation. This allows complex simulations to be built by simply choosing and connecting the desired components together.

Python Interface

While simulators [5] have been built around C++ and object-oriented techniques, Dshell++ wraps a Python [6] interface around the C++ classes so simulation setup and control can be completely script-driven. Selecting which models to include, specifying the data-flow connections and initializing the states and parameters are entirely specified through Python scripts that are processed at run-time. If a new, improved model becomes available, incorporating the new model is straightforward: simply swap out the old model with the new one in the Python scripts with no recompilation required. Users can access and even extend the simulation functions and C++ classes at run-time entirely in Python without modifying the C++ classes. Python provides all of the extensive features of a modern software language as well as additional ones including parsers, run-time loading of extensions, and a large collection of open source Python modules that are available to the simulation developers and users. Special functions, called “watch handlers” which are written in Python, can be created to trigger on events and monitor or plot data to the screen. Visualization in real-time (such as watching a rover slipping down a slope) is possible through the Dspace 3D toolkit [21]. Interfaces to external applications (such as Matlab) can be built entirely through the use of Python scripts.

Real-Time Performance Across Mission Domains

Dshell++ includes the multi-mission high-performance DARTS [13] flexible multibody dynamics module based on the Spatial Operator Algebra framework [8] for solving the dynamics of multi-body dynamics. Dshell++ has been used to develop real-time simulations for cruise/orbiter vehicles as well as to develop domain specific simulators such as ROAMS for surface rover simulations [9] and DSENDS [10] for entry, descent and landing simulations. These simulators

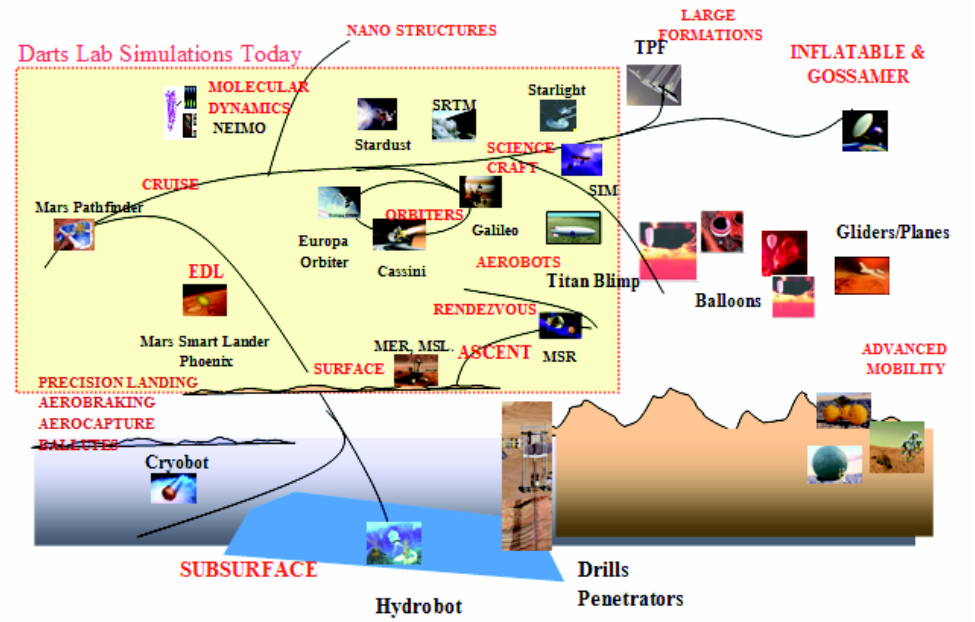

Fig. 1 Dshell++ has been used to create physics-based simulations for a broad range of mission domains.

have been in use by a number of successful NASA missions (Mars Science Laboratory, Phoenix) as well as technology development activities (Athlete [11] and the Lunar Surface Operations Simulator [12]).

Portability

Dshell++ is portable. The underlying framework is written in standard C++ and is highly optimized to run in real-time on laptops, desktop workstations and supercomputers.

II. Architecture

The key design requirements on Dshell++ are: reusability across mission domains, a modular design to reduce maintenance costs by simplifying updates and improvements, real-time performance and an interface which allows users to manage and customize complex simulations. We describe here the salient features of the Dshell++ architecture's that have been designed to meet these challenges.

A. Multi-domain Simulations

Physics-based simulations developed using Dshell++ are used extensively by space missions. Dshell++ based simulations have been used on workstations, in hardware-in-the-loop real-time simulations for space mission simulators for orbiter/cruise spacecraft, planetary surface rovers [12], entry-descent-landing simulators [10] and airship simulations (Fig. 1). Such simulations can include space environment models that are difficult if not impossible to create such as zero-g and planetary surface terrains. Such simulations are also used to explore a breadth of mission scenarios that would be too expensive or time-consuming to evaluate in physical simulators. Thus the family of simulations can vary widely across mission testbeds and from mission to mission.

As mentioned earlier, reusability across multi-domain simulations is a driver for component model-based simulation design. Complexity management is an equally important motivator for the modular architecture since hundreds to thousands of parameters and states are present in spacecraft simulators of even moderate complexity. Modularity brings with it the important benefit of encapsulation, information hiding and well defined interfaces across functional boundaries. Information hiding allows one to isolate functional blocks from each other and permit interactions only through clearly defined interfaces. It allows users to decouple and localize functionality within the simulation to facilitate testing, debugging, refactoring and evolution of simulations.

D. Data Flow

A data-flow architecture is used for simulation execution and component model interconnections and communications. Facilities are available to allow the exchange of typed data between models without inducing coupling between the details of individual model implementations. For example the output from a model generating information about the attitude and rate of a node can be distributed to other models needing this data through connector signals without exposing details on how the data are generated by the model (Fig. 2).

E. Model Organization

Beyond the data-flow modularity, component models are based on an object-oriented design where model classes can be organized into hierarchies involving related models, e.g. families of gravity models, thruster actuator models (Fig. 3).

Each Dshell++ model has a standard interface for parameter and state data as well as for flow inputs and outputs which can be customized for each specific model.

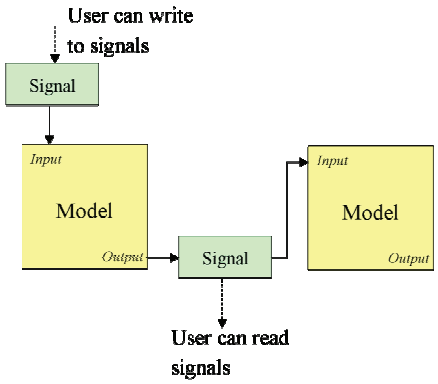

Fig.2 Data flow between models. In Dshell++, a model has user-defined *input* and *output* ports. Data is shared between models by tying a model's input port to another model's output port through a *signal* (basically a shared memory buffer). Users can peek or poke the signal data through the Dvar interface.

Model states can include both discrete and continuous states. A model interface file declares these externally visible characteristics for each model. Auto-code generators based on the Cheetah templating tool [16] were written to read these interface files and generate C++ model code.

The system's DARTS [13] multibody dynamics module is a key backbone layer supporting all the component models. The DARTS module is responsible for propagating the dynamics state of the system (e.g. spacecraft attitude and velocities, momentum, mobility slippage, vibration modes) using numerical integrators. The multibody dynamics states can be highly non-linear and coupled, and even though the physical assets (e.g. vehicles) may be distinct, their interactions must be handled together for the proper solution to the system's dynamics. As a consequence, the dynamics module is implemented as a unified model that includes contributions from all multibody components in the system. However, we have been careful to make sure this coupling does not adversely impact the overall modularity of the system. Towards this, component models have been designed to have a restricted interface to the multibody model to avoid unnecessary interactions and coupling among the component models.

F. Real-time Performance

Dshell++ simulators are used in real-time, hardware-in-the-loop, closed-loop simulations. High-performance speed is essential for such embedded use in time-critical testbeds. Toward this end, Dshell++ is designed to minimize unnecessary overhead that can impact performance. One potential area affecting performance is the need to exchange data among the several component models that constitute the simulation. Dshell++ provides a special connector facility called *signals* which allows the sharing of memory slots across models. Model outputs write to these shared memory slots while model inputs read from them. Thus there is no packetizing overhead from such data exchange.

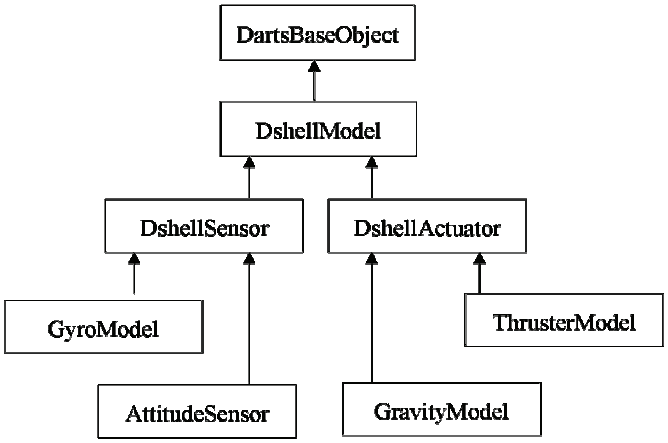

Fig. 3 Component models are based on an object-oriented design where model classes are organized into class hierarchies.

Many of the component models have to interact with the DARTS multibody dynamics [13] library which computes the kinematics and dynamics state of the space vehicle. The Dshell++ models have explicit interfaces to the relevant nodes and hinges in the DARTS model that allow them to make direct function calls to get/set the needed data efficiently.

The multibody dynamics module often dominates the computational cost for physics-based simulations. To address this, Dshell++ makes use of the DARTS dynamics engine (Fig. 4) that implements the fast Spatial Operator Algebra [8] based dynamics algorithms. The computational complexity of these algorithms is just linear in the number of degrees of freedom in the system. Moreover, DARTS allows the modeling of the dynamics of both rigid and flexible bodies with full implementation of the non-linear rigid-flex coupling to support very high-fidelity modeling with the most efficient algorithms available. DARTS' high-speed algorithms provide sufficient performance to typically allow the simulations to be used in mission testbeds without compromising performance speed or fidelity.

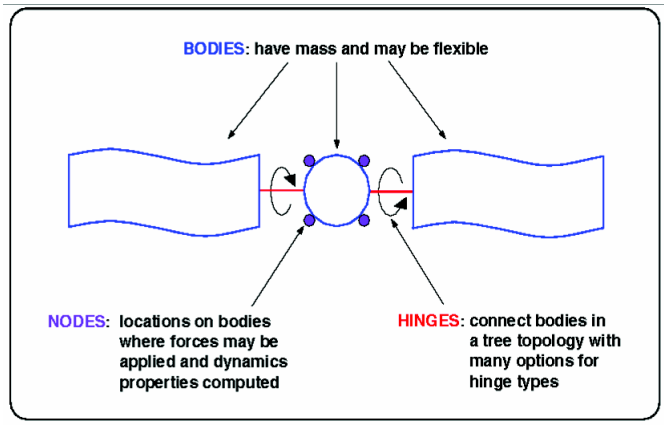

Fig. 4 The DARTS C++ library solves equations of motion for flexible multibody systems based on the dynamics properties of the bodies and the forces applied to those bodies.

G. Simulation Management

Simulator complexity is a natural byproduct of the large number of models involved in assembling typical spacecraft simulations. One of the goals of the Dshell++ architecture is to help manage this complexity. The approach is to build in checks that verify the consistency and correctness of the configured models in the assembled simulator. For instance, Dshell++ includes methods to verify that all model inputs and outputs are properly connected with matching types, which helps to reduce user errors.

Changes to the simulator configuration require explicit "unlocking" and "locking" of the system. When the system is "locked," any change to the topology of the simulation will be reported as an error. This ensures that simulation configuration changes cannot be made inadvertently and that appropriate checks and updates are made after the simulation's configuration has been changed. Examples of simulation configuration changes include the addition and deletion of models, multibody system bodies and nodes etc.

Services such as data logging, check pointing, data peeking and poking require the ability to interact with the disparate models to access the data specific to each model. While individual models provide APIs that allow users to interact with their internal data, the Dshell++ simulation architecture provides additional methods to access the overall data across all the models in the simulation.

While models can have discrete and continuous states, the latter require special handling since they are often coupled with the multibody system's dynamics state. The continuous states are propagated by numerical integrators such as CVODE [22]. Since numerical integrators typically work with contiguous memory blocks for states and state derivatives, Dshell++ takes care of mapping the individual model continuous state memory pools into contiguous memory blocks for interfacing with integrators. This bookkeeping is transparent to users and facilitates the use of different types of numerical integrators within the simulation.

Due to the data-flow architecture, it is important that the calling order sequence for the models is in accordance with their connectivity, i.e. models whose outputs are connected to the input of another model should be called before the dependent model. Since manually ensuring this requirement can be difficult and error prone once the number of models and interconnects exceeds even a small number, the Dshell++ architecture provides model order sorting facilities that process the model connectivity information to automatically determine the proper calling sequence for the models. Often times, the model inter-connectivity may result in connection loops among the models. Dshell++ provides methods to identify such loops and allow the user to define "breaks" in the loops to assist the sorting process. Dshell++ also allows a user to add extra dependencies into the sorting process over and above those implied by the model inter-connections. The model sorting process allows users to fairly easily build up simulations with hundreds of inter-connected models while enforcing the correct model calling order necessary for the correct execution of the simulation.

The modular implementation and encapsulation of the complex multibody dynamics model described above also goes a long way towards reducing the apparent complexity of the simulation by minimizing the coupling among the component models and layers.

H. Python Interface

While the Dshell++ library is written in C++, an extensive Python [6] scripting layer interface is also provided to its C++ classes and methods. The purpose of the Python interface is to allow the user to initialize and configure the simulation through a convenient scripting layer. The Python interface provides access to virtually all of the C++ methods in the Dshell++ classes. Indeed, the user has the option of setting up and configuring the full simulation in C++ or to do so entirely using Python commands (Fig. 5). A benefit of Python is the vast collection of open-source Python extension modules, e.g. socket programming, XML-RPC, and graphic widgets etc. can be easily used to extend the simulation capability in powerful ways.

```
from Dshell.Dshell import DshellX
from Dutils import Dvar_Py
DshellObj = DshellX(); # create Dshell object
execfile('model.py'); # load models
# Use Dvar to access the battery level parameter
batteryLevel =
  Dvar_Py.getDvar('.Spacecraft.DefaultSC.signals.batteryPowerLevel')
print batteryLevel();  # display the battery level at time t=0
DshellObj.step(10); # advance simulation by 10 seconds
print batteryLevel();# display the battery level at time t=10
```

Fig. 5 Sample Python script to run a Dshell++ simulation.

The low-level Dshell++ classes are all written in C++ for speed and execution efficiency. To generate the Python interface, we use SWIG (A Simplified Wrapper and Interface Generator) [15]. SWIG has the ability to generate a Python interface given only the C++ header (.h) files. The output from SWIG is a C++ file which is linked with the Dshell++ C++ libraries . Since SWIG only requires the declarations of C or C++ functions and classes and not the source code, SWIG can be used to generate the Python interfaces for third party libraries whose source code may be unavailable. With SWIG, a C++ program which calls Dshell++ can be rewritten entirely in Python. Users will only need to be familiar with Python to use Dshell++.

I. Graphical User Interface

While Python provides a command line interface for accessing simulation variables, it also forms the basis for auto-generating GUI panels to provide a more graphical and user-friendly interface for the simulation data. Dshell++ uses the GTK [17] widget family and its PyGTK Python binding [18] to build these graphical user interface panels at run-time (Fig. 6). These panels are tailored to the specific content of

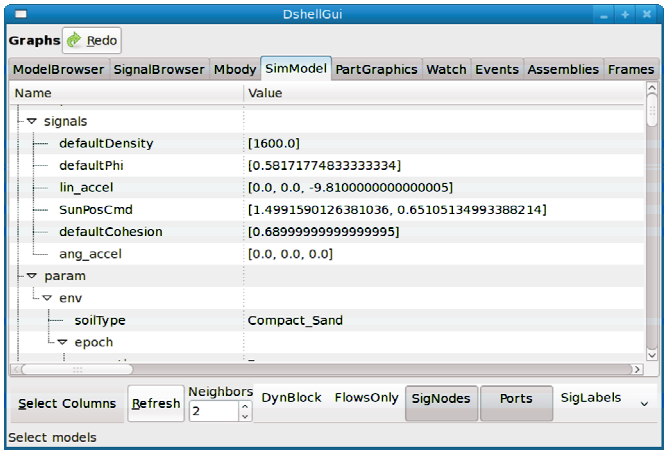

Fig. 6 The Dshell++ graphical user interface is built around the GTK and PyGTK graphics libraries and the Python scripting language.

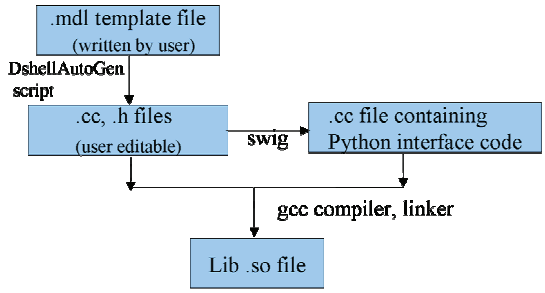

Fig. 7 Steps in building a Dshell++ model. You first create a model ".mdl" text file which describes the model's parameters, inputs and outputs. The DshellAutoGen (a python script based on the Cheetah templating tool) is used to create C++ .cc and .h skeleton code. You then edit the C++ files to add functionality to the model. The SWIG [15] tool is used to autogenerate a Python interface to the model. Finally, the C++ files are compiled to build a shared library (.so) file which can be imported in a Python script or linked to a C++ program.

the simulations being run. This is another example where generic features that adapt to the specific simulation model save the users the large cost and effort – and in this case the effort of building graphical user interfaces for their simulations. Additional support is available for creating strip chart panels to display the time histories of user-selected variables. These displays can be used to visualize the relative timing and values of variables as the simulation proceeds.

J. *Debugging*

The Python interface also provides powerful run-time methods to allow users to debug and tune simulations, such as the ability to activate and deactivate models while the simulation is running. This feature can be very useful for isolating problematic behavior in complex simulations involving tens or hundred of models. Another useful feature is the support for controlling multiple verbosity levels of debug messages. The verbosity of messages can be set to different threshold levels to filter debugging messages, warnings and errors. Moreover the fact that these settings can be set individually for models allows limiting of messages to just the relevant models and avoiding the torrent of messages that might otherwise be generated across the whole simulation. The built-in support for message sources and sinks allows the user to print messages directly to the screen, log to memory or to the file system for post processing. Another important issue in complex simulations is that of identifying and improving bottleneck models in a simulation. The Python date/time methods can be used for model profiling to collect the execution times of individual methods at the model level. This feature can be exercised interactively to identify areas of the simulation requiring additional tuning in order to improve simulation performance.

III. Simulation Development

A. *Generating C++ Component Model Classes*

To simplify building the C++ software for a component model, Dshell++ uses Python scripts to auto-generate the C++ class code for such new Dshell++ models. While the Dshell++ model base classes provide extensive functionality, additional methods are required to define the specific states, parameters and attributes of individual models. Much of this boiler-plate code comes from auto-code generation (Fig. 7).

Dshell++ uses the Python ConfigObj module [20] to parse Dshell++ model description (.mdl) files which describe the specific interfaces for a model (Fig. 8). The ConfigObj module provides classes to parse such data files. The extracted contents are used to auto-generate C++ code using the Cheetah [16] template-based code generator tool. The generated C++ code for the Dshell++ model contains stubs for the required methods for the model which the user can fill in with the model-specific functionality. Thus, the development of a new Dshell++ model reduces to one of creating the model interface definition file, running auto-code generator and then simply adding in a relatively small amount of model specific code. The auto-generated classes and code provide extensive functionality tailored to the model from which the user is freed the burden of writing. The auto-generation process is able to seamlessly handle changes to the model description files and merge in the auto-code updates with existing user defined code.

B. *Assemblies*

Since a simulation may contain hundreds of models, organizing the models can be a complex task. To address this issue Dshell++ implements an Assembly C++ class to build the sub-systems (Fig. 9). Assembly objects are containers and can contain models and even other assemblies. There can be multiple instances of an Assembly class. For example, you can instantiate four Wheel Assembly objects to represent the wheels of a four-wheeled rover. Assemblies simplify simulator design by allowing models and the complex interconnections to be grouped as a single package. The designer is then left with interconnecting assemblies together instead of delving in the details of the individual models.

```
[Type]
Name = Gyro

[flow_in]
  [[omega]]
  Type=double
  Length=3

[flow_in]
  [[accel]]
  Type=double
  Length=3

[flow_out]
  [[gyroCounts]]
  Type=int
  Length=3
```

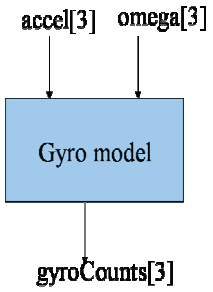

Fig. 8 Example of a model specification (.mdl) file. The user creates an .mdl file (a text file) to describe a model's inputs and outputs. For this example, the model simulates a hardware Gyroscope which has two inputs: the angular velocity (omega) and angular acceleration (accel). The output is an array of three integers which hold the gyro counts.

C. Configuration Files

Dshell++'s initial parameters are specified through Python scripts. The built-in support for object-oriented classes in Python supports rapid-prototyping and evaluation of new concepts directly in Python before migrating performance-critical ones to C++ eventually. An example of this is the use of Python extension methods in Dshell++ to process configuration files with model information to instantiate and build up the full simulation model. This process is much simpler than the alternative procedural process for setting up models. Moreover, the model configuration files are in Python syntax and are hence able to take full advantage of the Python interpreter's built in parser and its sophisticated error checking. The Dshell++ configuration files effortlessly support advanced features such as Python expressions, conditionals and even procedures, for example:

```
'SchackeltonCrater': {
  'Latitude': -1.5533, #radians
  'Longitude': 3.7175, #radians
  'Body': 'Moon',
  'SoilType': 'Compact_Sand'}
```

The advantage of using Python scripts is that parameters can be computed dynamically at run-time. For example, to specify the latitude/longitude in degrees instead of radians:

```
import math;
'SchackeltonCrater': {
  'Latitude': math.radians(-89.0),
  'Longitude': math.radians(213.0),
.  'Body': 'Moon',
    'SoilType': 'Compact_Sand'}
```

D. Signals

Dshell++ provides C++ base classes for hardware device models. These models share data through "signals" (Fig. 2).

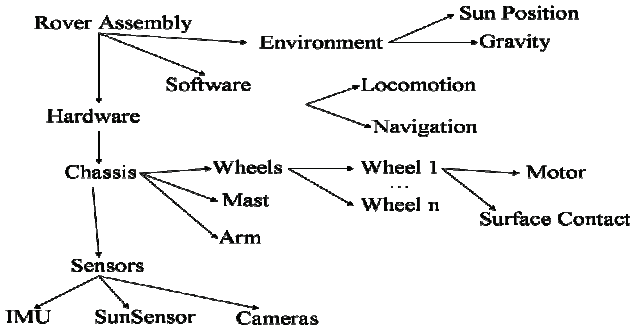

Fig. 9 A Rover Assembly Diagram. Related models within subsystems (e.g. a wheel motor, and surface contact model) can be grouped into a "wheel assembly." Assemblies are reusable and can be used in more than one simulation. This allows complex simulations to be built by choosing and connecting the desired assemblies together.

A model writes its output to one or more signals and reads input from signals. A signal is essentially a C++ array in memory. When a model writes to a signal, it writes to this memory location where it is available for another model to read. This shared memory paradigm allows a model to be designed with no C++ dependencies on other models. Model inputs and outputs are simple C++ pointers to the signal so changing the source code of a model does not require recompiling other models.

E. Dvar Interface

Several factors that go beyond model correctness and performance are critical to the usability and scalability of simulation architectures. The complexity of debugging, testing and validating of simulations grows exponentially as the number of component simulation models and their interconnections grow. As a consequence, the sustainable use of a simulation architecture for modest to large-size simulations depends critically on the level of built-in features that allow analysts and users to manage the growing complexity. Thus, Dshell++ includes several simulation features and services designed to meet these needs.

Since even modest scale simulations can involve thousands of variables, one of the pressing usability needs is to provide analysts with a way to peek and poke at simulation variables at run-time. Dshell++ includes a very flexible framework-level layer, called *Dvar* (**D**shell **var**iables), to support querying and modifying virtually all simulation variables interactively at run-time. Dshell++ implements Dvar C++ classes to represent the basic C types (float, short, bool, double, int, long, enums), strings and arrays. Simulation variables, such as those associated with component model inputs/outputs, parameters/states, or with multibody model states are organized into a Dvar namespace hierarchy. This hierarchy forms the basis of an addressing scheme which assigns to every variable a unique path-like string address in the Dvar variable space. Dvar's C++ or the Python interface can be used to locate and work with any variable in the simulation at run-time. In this approach, a user has unlimited

access to variables and is not constrained to a pre-defined set of variables. This is very powerful feature for debugging and test sessions where even the finest grain details can be monitored for debugging purposes.

Here's a C++ example of how Dvar works:

```
static double mass;  // this is the variable we want to register
Dvar::create("mass", &mass); // registers "mass"
//You can now access the variable through Dvar C++ methods
DvarDouble *p = Dvar::getDvarDouble("mass");
p->value(10.2);  // sets mass = 10.2
```

The C++ variables can also be accessed in Python scripts through Dvar:

```
# Python code
import Dvar_Py;
Dvar_Py.getDvar('mass')(10.2);  #  will set the C++ mass to 10.2
```

Conversely, one can create Dvar objects in Python to wrap Python objects and access those Python objects through C++ code.

Dvar objects can be also organized into a tree-like nested structure where a Dvar object can contain other Dvar objects. Here's a sample Dvar Python code to change the efficiency parameter of a solar panel model attached to a space vehicle named "Rover":

```
import Dvar_Py;
Dvar_Py.getDvar(
    '.Spacecraft.Rover.models.SolarPanel.params.Efficiency')(0.90);
```

F. Check Pointing

Since a Dvar tree can be saved to and read back from a file; Dshell++ uses the Dvar interface to check point and resume simulations. The check point function stores the full simulation state to the file system. The simulation state can be restored from such check point files. This allows users to restart simulations from arbitrary points in long simulation runs.

G. Events and Watch functions

The Dvar interface also supports a "watch" feature that allows a user to register callback functions for selected Dvar variables. The callbacks, which are functions written in either Python or C++, are triggered any time the associated variable's value changes. This allows the user to monitor variables and trigger events only when something interesting happens. Using Python scripts as the event and watch handlers instead of C++ functions has the advantage of allowing users to add or modify the handlers without the need for recompiling the source code.

One of the key users of this feature is Dshell++'s interface to its 3D graphics module called Dspace [21]. Watch handlers are attached to the Dvar simulation variables associated with the position and attitude variables for physical bodies in the simulation. When their values change, the handlers are triggered and send messages to Dspace to update the position and location of the graphics objects in the scene to keep them in sync with the simulation. Other uses of the watch feature are for updating displays and data logging.

While watch variables are tied to Dvar variables, a separate "events" module allows users to register callbacks that are automatically invoked as the simulation time advances. The rate of invocation can be set to be periodic with a user defined frequency or can be configured to be triggered after a specified delay.

H. Data Flow Visual Displays

The Graphviz [19] open-source library is used for auto-generated visual displays of the Dshell++ model data-flows and their interconnections (Fig. 10). Since these displays can become very dense, Dshell++ allows the user to interactively center the displays around any model and control the "neighborhood" of models to be displayed. Such displays can be used to verify that the models are interconnected properly as well as to document the simulation design.

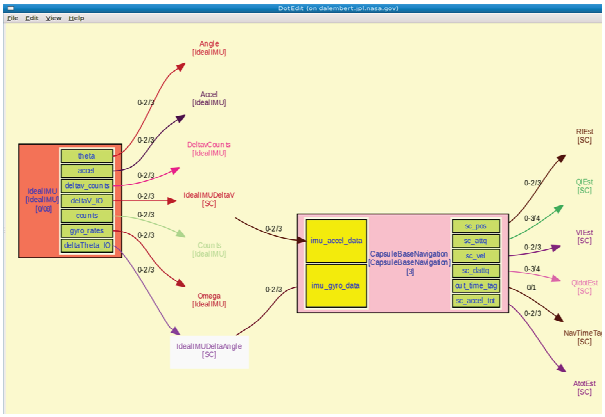

Fig. 10 A signal flow graph displaying the connections between models.

IV. Applications

Dshell++ based simulations have been used on workstations, in hardware-in-the-loop real-time simulations for space mission simulators for orbiter/cruise spacecraft, planetary surface rovers, entry-descent-landing simulators and airship simulations. Some examples are presented below.

A. Surface Rover Vehicles

Dshell++ has been used to build the ROAMS [9] physics-based, high fidelity simulator for planetary surface exploration rover vehicles. Rover components such as a stereo camera, navigation sensors and motor control are all modeled using Dshell++.

B. Entry, Descent and Landing

EDL simulations for the Mars Phoenix Mars Lander [14] are performed using the Dshell++-based Dsends [10] simulator for maneuver targeting, landing dispersion analysis and safety assessment. The simulation implements a 6-DOF

model of the entry trajectory, parachute deploy and lander descent to the surface of Mars.

C. Manned Lunar Operations

The Lunar Surface Operations Simulator [12] has been developed with Dshell++ to support planning and design of future manned missions to the moon. Various lunar rovers, habitats, dynamic and physical processes, and environment systems are being modeled and simulated.

V. Conclusion

Dshell++ makes full use of object-oriented techniques to allow code reuse and component building to minimize development and maintenance costs. Performance-sensitive code is written entirely in C++ and a Python scripting interface is used for the simulation configuration and user interaction. Python's object-oriented paradigm provides an excellent match to the Dshell++ architecture and reduces new development costs by allowing Dshell++ simulations to interface with the huge number of off-the-shelf Python libraries.

Future plans include adding a multi-rate scheduler to support models which require high sampling rates, the ability to run each model in a separate thread, and efficient data loggers to capture results to a database.

VI. Acknowledgment

The authors would like to express their thanks for the invaluable discussions with our team members at the Jet Propulsion Laboratory's Dynamics and Real-Time Simulation Lab (DARTS Lab): J. (Bob) Balaram, Jonathan Cameron, Rudranarayan Mukherjee, Hari Nayar, Marc Pomerantz,, and Leonard Reder. Special thanks to Todd Litwin for reviewing the paper. The research in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

[1] G. Braun, D. Cornick, R. Stevenson, "Capabilities and Applications of the Program to Optimize Simulated Trajectories (POST)," in *NASA CR-2770*, Feb. 1977.

[2] Nemeth, S., "Hybrid Simulation Technology: The Next Step in the Evolution of Spaceflight Simulations," in *SpaceOps 2008 Conference*, Heidelberg, Germany, May 2008.

[3] "Matlab/Simulink Web Site," URL: http://www.mathworks.com.

[4] J. Biesiadecki, D. Henriquez, and A. Jain, "A reusable, real-time, spacecraft dynamics simulator," in *6th Digital Avionics Systems Conference*, Irvine, CA, Oct. 1997.

[5] *SMP 2.0 Handbook*, European Space Agency EGOS-SIM-GEN-TN-0099, Oct. 2005.

[6] "The Python Programming Language,". URL: http://www.python.org

[7] A. Jain, J. Guineau, C. Lim, W. Lincoln, M. Pomerantz, G. Sohl, R. Steele, "Roams: Planetary surface rover simulation environment," in *International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS 2003)* Nara, Japan, May 2003.

[8] G. Rodriguez, K. Kreutz-Delgado, A. Jain, "A Spatial Operator Algebra for Manipulator Modeling and Control," in *The International Journal of Robotics Research*, vol. 10, pp. 371-381, Aug. 1991.

[9] A. Jain, J. Balaram, J. Cameron, J. Guineau, C. Lim, M. Pomerantz, G. Sohl, "Recent Developments in the ROAMS Planetary Rover Simulation Environment" in *IEEE 2004 Aerospace Conference,* Big Sky, Montana, March 2004.

[10] J. Balaram, R. Austin, P. Banerjee, T. Bentley, D. Henriquez, B. Martin, E. McMahon, G. Sohl, "DSENDS - A High-Fidelity Dynamics and Spacecraft Simulator for Entry, Descent and Surface Landing," in *IEEE 2002 Aerospace Conference*, Big Sky, Montana, March 2002.

[11] B. Wilcox, T. Litwin, J Biesiadecki, J. Matthews, M. Heverly, J. Morrison, J. Townsend, N. Ahmed, A.Sirota, B. Cooper, "ATHLETE: A Cargo Handling and Manipulation Robot for the Moon," in *Journal of Field Robotics 24(5),* April 2007.

[12] H. Nayar, J. Balaram, J. Cameron, A. Jain, C. Lim, R. Mukherjee, S. Peters, M. Pomerantz, L. Reder, P. Shakkottai, S. Wall, "A Lunar Surface Operations Simulator," in *Proc. International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008),* Venice, Italy, Nov. 2008.

[13] A. Jain and G. Man, "Real-time simulation of the Cassini spacecraft using DARTS: functional capabilities and the spatial algebra algorithm," in *5th Annual Conference on Aerospace Computational Control*, Jet Propulsion Laboratory, Pasadena, CA Aug. 1992.

[14] E. Bonfiglio, D. Adams, L. Craig, D. Spencer, W. Strauss, F. Seelos, K. Seelos, R. Arvidson, T. Heet, "Landing Site Dispersion Analysis and Statistical Assessment for the Mars Phoenix Lander," in *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, Honolulu, Hawaii, Aug. 2008.

[15] "A Simplified Wrapper and Interface Generator (SWIG)," URL: http://www.swig.org

[16] "Cheetah, A Python-Powered Template Engine," URL: http://www.Cheetahtemplate.org

[17] "The GTK+ Project," URL: http://www.gtk.org.

[18] "PyGTK: GTK+ for Python," URL: http://www.pygtk.org.

[19] "Graphviz –Graph Visualization Software," URL: http://www.graphiz.org

[20] M. Foord and N. Larosa, "ConfigObj," in http://www.voidspace.org.uk/python/index.shtml.

[21] M. Pomerantz, A. Jain, "Dspace: Real-Time 3D Visualization System for Spacecraft Dynamics Simulation," in SMC-IT 2009, Pasadena, CA, July 2009.

[22] A. Hindmarsh, R. Seran, "CVODE," in https://computation.llnl.gov/casc/sundials.